An Analysis of the Application of Ant Colony Optimisation and Genetic Algorithms to the Learning of Successful Combinations of Strategies in Robocode

Max Ward Mitchell Misich

November 4, 2013

Abstract

Table of Contents

1 Introduction

The goal we selected for this project was to create a system which could learn to win one versus one battles in Robocode. This is an ambitious, and rather holistic goal. We did not want to learn only a small or restricted part of the Robocode game, instead we liked the idea of being able to have a robot learn (from the ground up) how to win against an opponent, or at least how to lose in style.

The first approach that we implemented was a flavour of Neuroevolution. This required implementing both a neural network, and a genetic algorithm to train said neural network. We then attempted to use Neuroevolution to learn how best to control a robot from scratch; this can be described as a holistic approach. Our second attempt was based on Ant Colony Optimization (ACO). It was less ambitions than our first approach, and attempted to come up with the best combination of pre-designed, modular strategies to defeat an opponent. In addition, we also applied Neuroevolution to fine tune ACO, once it found a good strategy. In order to compare the performance of ACO against another method to analyse its ability to learn such a strategy, a genetic algorithm was also used.

1.1 Ant Colony Optimization

ACO is a metaheuristic inspired on the foraging behaviour of Argentine ants. Our ACO is based loosely on the Max-Min Ant System in [1]. We keep a finite population of ants which walk through a graph representing the solution space of the problem. The way these ants choose nodes is the first important feature that makes ACO an effective algorithm. The second is the simulation of ants laying down pheromone.

1.2 Genetic Algorithms

Genetic algorithms as defined in [2] are a class of evolutionary algorithms which use a population of collections of descriptors which describe how they are characterised in the problem domain. As such each descriptor is often called a gene, a collection of genes is often called a genome or genotype, and the characterisation of the genotype in the problem domain is called a phenotype. Every member of the population has a 'fitness' which is used to find optimal solutions in the problem domain. Phenotypes which are more optimal will have a greater fitness. Members of the population interact using genetic operators such as mutation and crossover to produce new members which may or may not be more optimal, i.e. have greater fitness. The rate at which these operators occur balance exploitation vs. exploration.

Genetic algorithms are used in two different ways in this project. The first is to train artificial neural nets in Neuroevolution, and to learn the best combination of predefined strategies to combat an enemy. The other application is to use genetic algorithms to learn the best combination of predefined strategies to combat an enemy in contrast to Ant Colony Optimisation.

There are many different types of genes. Common gene representations are bits, integers, and floating-point numbers. These can be used to represent many different properties; such as membership, a class, or a weight in a graph (respectively). Therefore an entire network can be represented by a genotype of a collection of floating-point numbers, where each floating point number is a weight on an edge, where the particular edge is identified by the location of the floating-point number in the genotype.

When using a genetic algorithm, the user needs to define genes, genotypes, phenotypes and a fitness which describes how fit each phenotype is in the problem domain.

At each iteration of the algorithm in which the population is evolved genetic operators are applied. The genetic operators used are:

- Selection;
- Elitism;
- Crossover; and
- Mutation.

Elitism is used in order to increase the rate at which the algorithm converges. It works by taking a fixed percentage of the population which are the fittest members into the next generation. This also ensures that the fittest solution found at any iteration is in the final population. It improves the rate of covergence because the elite members are more likely to be chosen in future iterations for crossover and mutation; and so the genetic information from the elite members always survives and can spread to new members. Ideally, the genes which are transferred to new members are responsible for their high fitness and so new members also have similar fitness as the elite members; thus converging quickly.

Selection is a method for selecting members for the crossover and mutation operators. Two types are investigated, Tournament and Truncation selection.

Selection methods do not produce new members but they are used to select members for crossover and mutation. A genetic algorithm exploits the current population by tending to select members which already have a high fitness. Therefore selection operators tend to select fitter members. It is also important to increase diversity and exploration. This is done by probabilistically selecting members such that fitter members are more likely to be selected, but weaker members also have a chance to be selected. Tournament selection allows this because the fittest member selected for the small tournament may not be the same as the fittest member of the population. Small tournament sizes favour exploration; increasing divesity.

The other selection operator used was Truncation selection. Truncation selection takes a certain percentage (e.g. 40%) of the fittest members of the population as potential parents. Members are selected uniformly with replacement from the truncated population for both mutation and crossover. The uniform selection allows the weaker members of the truncated population to be selected increasing diversity; but the truncated population eliminiates potential very bad choices increasing the rate of convergence.

Mutation is an operator which allows for new genetic information which may not already exist within the population to be introduced into the population. Mutation works by randomly changing a collection of genes within a genotype. There are different types of mutation, for example reselection of a gene (uniformly or by another probability distribution); scaling a gene by some amount especially if it is a number; or a random walk in a graph that has been imposed on the potential genes. As such new genes can be selected which may improve the current fitness. Mutation is required because it allows genetic algorithms to escape local optima. If the algorithm converges prematurely because of a lack of genetic diversity, then mutation can increase the amount of genetic diversity allowing the genetic algorithm to further explore the problem domain.

Crossover (or recombination) is an operator which combines two (or more) selected members of the population into a collection of 'offspring' which use genes from the 'parents'. There are many different types of crossover operations. A common crossover operation for strings of genes is one-point crossover where a point is uniformly chosen in the genotype and two children are created by using the first half and the second half of each parent. Multiple crossover points can also be used. If a genetic algorithm lacks genetic diversity then they converge to a local optima because there isn't enough distinctive genes to change the fitness and the genotypes of the members of the population. Methods are often implemented to guarantee a specific level of diversity. Such methods are related multimodal optimization where the population does not focus toward a single optima, rather, tries to sustain many optima.

1.3 Neuroevolution

Neuroevolution is a population-based technique for learning the weights of an artificial neural network. The Neuroevolution method used is a direct representation. This means the genomes in the genetic algorithm map directly to the neural network. Our genomes were simply a vector of real numbers which represent the weights of the neural network. This straightforward representation has the advantage of being easy to code and easy to train. This technique, however, only works with a good initial network topology because the genetic algorithm cannot change the structure of the neural network.

2 Implementation

2.1 Implemented Strategies

2.1.1 Abstraction

A level of abstraction was implemented between the ACO and GA methods of finding an optimal combination, and the evaluation of fitness of the combination. This is because the genome of the ants and the genotypes in the GA are the same quadruples of strategies. This allowed for more code to be reused. Both GA and ACO produce candidate solutions which are abstracted using an interface called `StrategySolution`, and the GA and ACO methods implement a `StrategyOptimizer` interface.

Movement	ID 0: CircleMovement, ID 1: TightCircleMovement, ID 2: BigCircleMovement, ID 3: SpiralMovement, ID 4: RandomAngleMovement, ID 5: BackwardCircleMovement
Firing	ID 0: WeakFire, ID 1 StrongFire, ID 2: CloseFire, ID 3: FireIfSlow, ID 4: MediumFire, ID 5: MaxStrengthFire
Scanning	ID 0: MatchSlowRobotScan (matches the robot's heading, slowly), ID 1: FastSpinScan, ID 2: SlowPredictiveScan (tries to get a scan lock, slow moving), ID 3: SlowSpinScan
Gun Movement	ID 0: SlowFollowScannerGun, ID 1: FastFollowScannerGun, ID 2: SlowSpinGun, ID 3: FastSpinGun, ID 4: FollowEnemyGun

Table 1: A Summary of the Strategies Combined in to form the Solution-Space of the Ant Colony Optimization and Genetic Algorithms

The robot which implements each solution for the purposes of calculating fitness only uses the `StrategySolution` interface which supports a method for returning the solution by a method call `getSolution()` which returns an integer array describing the selected strategies. This array contains exactly four values. Each value identifies a specific strategy. First first number in the array defines a movement strategy, the second defines a firing strategy, the third defines a radar movement strategy, and the last defines a gun movement strategy. Every strategy was simply a pre-programmed function which took as input the robots current knowledge and returned the next set of actions the robot should perform. The solution space contained all possible combinations of strategies for each of the four aforementioned classes. The solution classes were programmed to be completely modular; movement strategies only controlled the robot's movement, radar movement strategies only controlled the radars movement, et cetera. Clearly this is the same process as that used by Hong and Cho [3].

We have, however, made some modifications to their method. Firstly, we allowed a strategy to access all the data the robot is allowed to access. We also removed the target selection strategy as we are only focussing on one versus one battles. We also removed the 'avoid' strategies. A summary of the strategies we implemented can be found in Table 1.

2.1.2 Measuring the Fitness of a Strategy

The fitness of a combination of strategies is measured by performing 40 rounds in Robocode with the predefined enemy robot and observing the ratio of the robot's score to the enemy's score.

$$\text{Fitness}\,(phenotype|enemy) = \frac{S(p)}{S(p) + S(e)} \tag{1}$$

The score ratio is an excellent fitness measure because it indicates how the robot performs against the enemy and how much it 'wins' in proportion to the enemy. It is better than just using the total score because even though the robot may win, it could have received a low score because it won quickly or won by dodging the enemy's bullets and firing none of its own. Using the score ratio abstracted these fluctuations.

In addition, using 40 rounds removes the variation caused by random starting locations. Some locations may give the robot an unfair starting advantage. Therefore a large number of rounds is used to accurately compute a good fitness.

2.2 Ant Colony Optimization for Strategy Combination

Initially, an ant will compute how much interest it has in a node. This is described in Equation 2.

$$\text{Interest} = r \times m + p \tag{2}$$

where the variable r is sampled uniformly from $[0, 1)$ and the variable m is a randomness factor chosen by the programmer. In our implementation this was set to 1.0 by default. Finally, p is defined as the amount of pheromone on a node. The Max-Min Ant system constrains the value of p to $[0, 2]$, and so does our algorithm [1].

The ant then walks to the node with the maximum interest. It continues to do this until it has reached maximum depth. After all ants have walked through the graph, the amount of pheromone on each node of the graph decreases by a small amount (chosen to be 0.05 in our implementation).

The fitness is calculated for all of the "ant walks" and the fittest ant is allowed to lay down some new pheromone. It does this by putting down Q/L pheromone on each node where Q is the total amount of pheromone an ant carries (chosen to be 1.0) and L is the number of nodes in the solution.

2.3 Genetic Algorithm for Strategy Combination

The genetic algorithm uses the same candidate solutions as the ants in the ACO implementation. It implements the abstracted `StrategyOptimizer` interface.

2.3.1 Gene Representation

If strategies are being combined and each strategy belongs to a particular class, then each strategy in each class can be assigned arbitrarily a unique integer identifier. These identifiers become the genes. Each gene is an integer describing which member within a particular class of strategies to use.

2.3.2 Genotype Definition

A collection of each of the class identifiers or integers describes is the genotype. In this application, a genotype is an ordered quadruple of integers. The first integer describes

the movement strategy, the second describes the fire strategy, the third describes the scan strategy, and the fourth describes the gun movement strategy.

2.3.3 Phenotype Definition

A phenotype is therefore the actual robot which uses each of the strategies defined in the genotype. The fitness of a phenotype is given by Equation 1.

The selection method used is Tournament selection. The version of tournament selection used here is to select a 3 members uniformly with replacement and take the fittest.

2.3.4 Mutation Operator

The mutation operator which is used in this application is random reselection of a number of genes. The number of genes to mutate in a given genotype is modeled by a Poisson distribution with an expected number of mutations, $\lambda = 1$. Therefore, probabilistically, it is possible the selected individual will not be mutated to favour exploitation, or may have many (even greater than 1) mutations favouring exploration.

2.3.5 Crossover Operator

The method used in this application is the one-point crossover. It is hoped this would allow combinations which have a good gun-scan strategy would be able to combine with a good combination of movement-fire.

2.3.6 Maintaining Diversity

A simple method to achieve diversity in the population is to ensure members cannot be clones of each other. The genotype space is fairly small and therefore there is enough genetic diversity if all members are distinct.

2.4 Robot Training using Neuroevolution

Several activation functions in the neural network were investigated. These include a simple threshold function; `tanh`, and the logistic `sigmoid` function. The threshold function was not ideal as it prevented Neuroevolution from benefiting from small changes in weights;– there was no gradient of error. There was no significant difference in fitness between using the `tanh` and `sigmoid` function; but the implementation of `tanh` in Java was mysteriously very slow and as such a logistic `sigmoid` ANN was used. A post on `StackOverflow` [4] would seem to indicate this is expected and that Java's math library stressed correctness over performance.

A pitfall of our Neuroevolution style is it suffers from competing conventions. This is when a population contains two or more different encodings for the neural network which produce similar outputs. This confounds the genetic algorithm somewhat because crossover will generally produce offspring with low fitness [5].

In addition to this, Neuroevolution like back-propagation may become trapped in local minima, particularly if there is not enough variation in its starting population. It may also become trapped if suboptimal genomes initially have quickly increasing fitness; but plateau soon after.

To address this problem we use isolated populations in the genetic algorithm. For every generation two randomly selected populations trade their worst fitness individual for a copy of the other population's best. This promotes diversity; but ensures good innovations eventually spread throughout the individual populations. A similar result might be achieved using niching; however our approach was easier to implement.

2.5 Neuroevolution for Tuning Strategies

In our ACO and GA algorithms we defined how the robot should fire and move its gun but never when to fire its gun. When training the combination of strategies we used a default strategy which was quite effective.

Whenever a robot was scanned, we saved its (X, Y) coordinates. In the main loop when deciding whether to fire or not, we calculated the angle between the enemy's most recently scanned location and robot's current location. The difference Δ between this angle and our robot's current gun heading is used to decide whether or not to shoot. The default policy is to shoot iff $-10 \leq \Delta \leq 10$, and if the last scan was less than 5 turns old.

After learning a good strategy using the ACO or GA methods we applied Neuroevolution to learn the fire function. The inputs were Δ, and the age in turns of the last scan. The output was a boolean: fire or don't fire. The same implementation of Neuroevolution described in the previous section was used with the network topology detailed in Figure 1.

2.6 Fitness Calculation and Issues Faced with the Robocode API

2.6.1 Strategy Learning

Fitness calculations were performed for each candidate `StrategySolution` by writing the selected strategies to file. A generalised robot class `ScriptBot` was created which reads the strategies to file, creates instances of the selected strategies using a `StrategyFactory` class, and executes them when required by the Robocode environment.

Each battle is handled by an instance of an implementation of the Robocode `BattleAdaptor` called `BattleRunner`. Only one instance of `BattleRunner` is used throughout all the fitness calculations for every generation. Each instance creates an instance of the `RobocodeEngine`. Initializing the `RobocodeEngine` takes time; and therefore by only constructing one instance time is saved.

Fitnesses were calculated sparingly. Elite members carried to the next generations did not have their fitnesses calculated. Further, in the GA, mutant genotypes and crossed-over offspring only have their fitnesses calculated if they are not the clone of a member already in the next population.

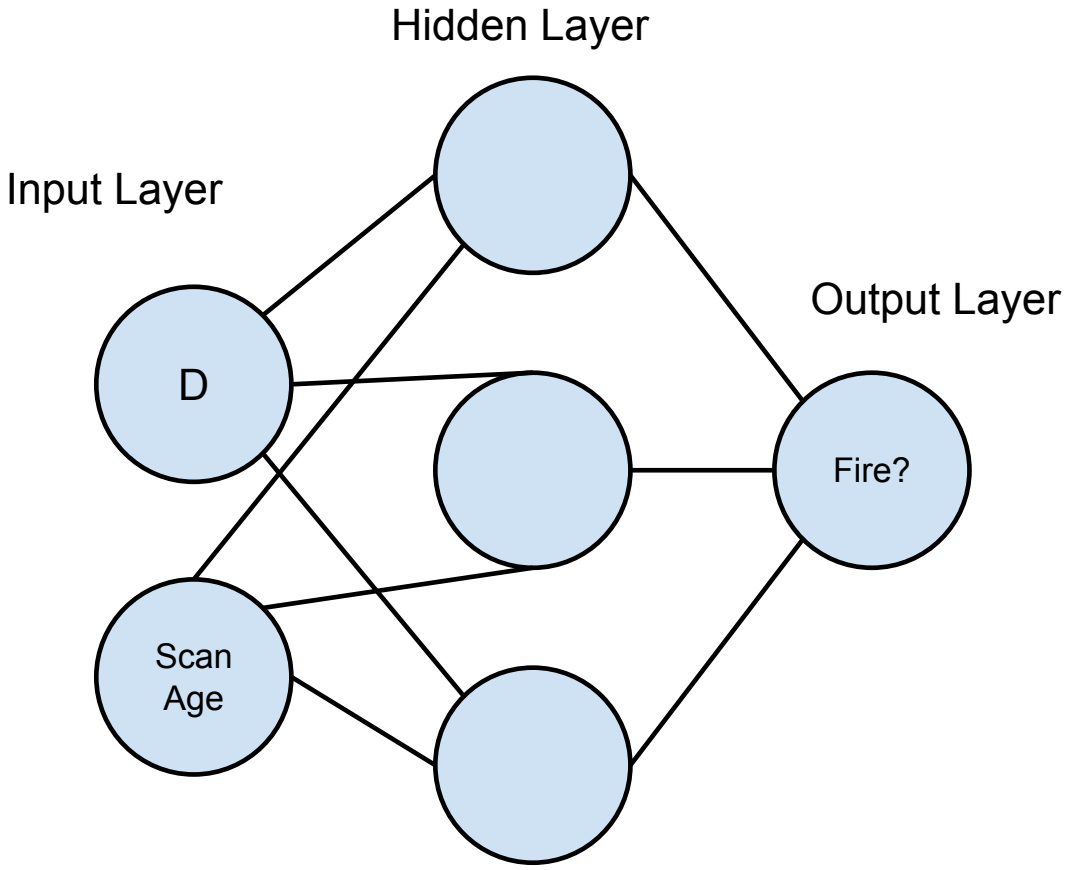

Figure 1: An Artificial Neural Network Topology for Neuroevolution for Tuning a Firing Strategy

In order to tune the best found strategy, a different class `NNScriptBot` was used which not only loads the strategies akin to `ScriptBot`; but also loads a neural network specification which describes the topology of the neural network and the weights of each of the corresponding edges.

The `BattleRunner` class when executing a battle passes in a string representing the class name of the enemy, for example "`sample.Tracker`" and the string representing the corresponding script bot, "`sample.ScriptBot`".

2.6.2 Robot Training using Neuroevolution

The fitnesses of robots trained using Neuroevolution were computed in a similar manner to `NNScriptBot`; using a class called `NNBot`. A specification was loaded and executed.

2.6.3 Issues Faced using the Robocode API

Unfortunately the Robocode API cannot accept instances of `Robots` in a "`runBattle`" function. The File input and output described above was used as a work around to specify learned behaviours. This disk I/O turned out to not be such a bottleneck for learning; but it did make the fitness calculations more awkward to compute. As such they were abstracted by a `GeneralFitness` interface which takes in a candidate and outputs the score ratio.

3 Experiment Design

3.1 Robot Training using Neuroevolution

3.1.1 Aim

The aim of this experiment is to test the Neuroevolution implementation and to determine if it is appropriate for learning a general Robocode strategy from scratch.

3.1.2 Experimental Hypothesis

It is hypothesised using a pure holistic Neuroevolution approach to learning a general strategy for a Robocode robot is achievable.

3.1.3 Methodology and Rationale

Neural Network Topology

Neuroevolution requires a specified topology of the network. Preliminary experimentation revealed the topology defined by the vector $< 13, 15, 7, 5 >$ was effective for learning without taking too long to learn. There are a total of $13 \times 15 + 15 \times 7 + 7 \times 5 = 335$ weights. It is expected such a topology allows for the inputs to be expanded upon and then combined slowly to the outputs.

Number of Populations and Population Size
: Neuroevolution was done using 3 populations containing 32 individuals each. Evaluating the fitness is the most significant time expenditure and therefore the total number of individuals is kept fairly small.

Running the Algorithm
: The algorithm was run overnight because it ran so slowly, and the presented results show fitness at every tenth generation to help remove noise.

Fitness
: Unless otherwise stated, the fitness function used was the same as Equation 1. The score is the total score after 40 rounds.

Neuroevolution Genetic Algorithm Parameters
: The parameters set for the GA used for training were found by trial and error. A truncation factor of 50% was used. Anything less tended to mean that the algorithm converged on suboptimal solutions too quickly. We had the crossover rate was set to 35% and the mutation rate was set to 60%, and therefore a 5% chance of adding a new random individual in place of breeding an old one.

  The crossover weight was attempted to be set higher; however this lead to much slower rates of convergence. Setting it too low also presented the same problem. This is likely due to good innovations – which can be shared between different populations – were never able to mix with other good but incomplete solutions.

  Having a small chance of introducing new individuals made the average fitness appear more noisy, but also seemed to prevent premature convergence.

3.2 Strategy Learning using ACO and GA

3.2.1 Aim

The aim of this experiment is to investigate the difference in performance between the use of Ant Colony Optimization and Genetic Algorithms for the purposes of optimizing the fitness of a combination of strategies from a finite set of predefined Robocode strategies.

3.2.2 Experimental Hypothesis

It is hypothesised both methods will be able to learn winning strategies against fixed enemy robots.

It is also expected the genetic algorithm will achieve better fitnesses than the ant colony optimization because it is well understood and has been applied before to similar solution spaces.

It is also expected that Neurotuning will increase the performance of the robots learned by the ACO or the GA.

3.2.3 Methodology and Rationale

Population Size for ACO
: In all the tests, a population of 32 ants run for 25 generations was used. These numbers seem small but preliminary investigations showed it is all that is required for finding excellent solutions. This finding is echoed in our results.

Population Size for GA
: In order to compare the GA against the ACO algorithm, 32 genotypes were used in a single population and the GA was also run for 25 generations. It is hoped this allows for a fair comparison between the two methods.

Fitness
: As before, the fitness is measured in accordance with Equation 1 over a 40 round period.

Tuning the Fire Strategy using Neuroevolution
: When tuning the firing function, the weights of the neural network were configured to use 2 populations with 16 genomes per population for 25 generations. The best found strategy from both GA and ACO were used to control the robot; albeit with the evolved neural net determining its 'to fire' function.

  Again the results showed this was all that was required to get excellent results.

Enemy Robots
: The results shown use two sample robots from the Robocode library; RamFire and Tracker. These robots were chosen because they are two of the hardest robots to beat, and if the learned strategies are able to beat them then it is enough to illustrate the efficacy of our techniques.

4 Results

4.1 Robot Training using Neuroevolution

4.1.1 Rate of Convergence

Pure Neuroevolution was never able to learn how to beat any of the sample enemies provided as part of the Robocode. It never came close, even after hours of learning. Though not heartening, this was after all not the aim of our investigation. The important question for us was whether or not Neuroevolution was able to learn and thereby improve its performance. The answer is a tentative yes.

Figures 2 and 3 show while the fitness of our implementation of Neuroevolution seems to learn at a slow but noticeable rate it tends to slow down after a large number of generations.

The elitism in the genetic algorithm for training the network leads to the plateaus seen in Figures 2 and 3. This is because the fitnesses of these individuals were not recal-

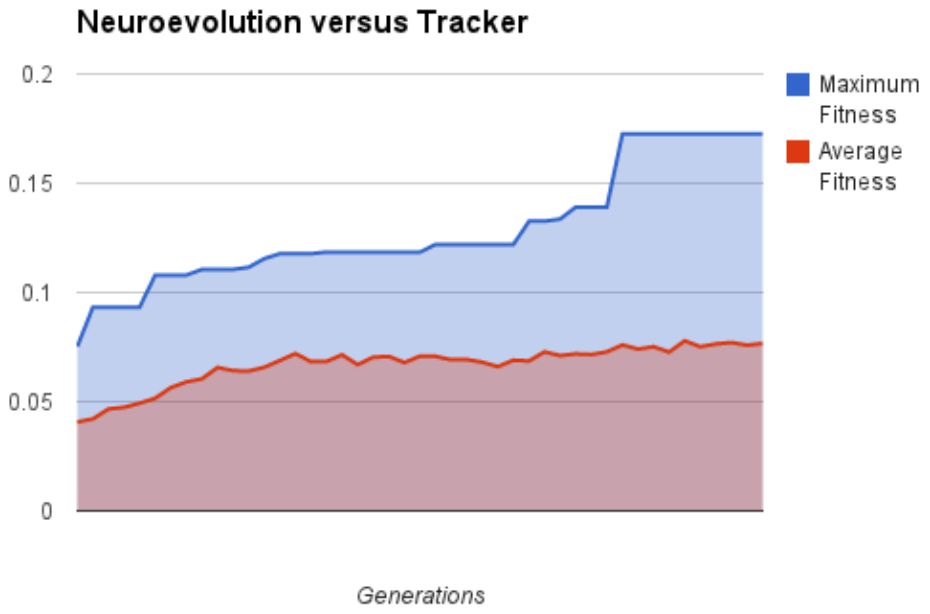

Figure 2: Learning a General Strategy using Neuroevolution against Tracker

culated every generation to avoid longer training times. Every genome was immutable so it was assumed the fitness is the same even though probabilistically it may change.

4.1.2 Observed Behaviours

During the first generation, most strategies looked like a random walk. Sometimes, however, a Robot would spin its radar, gun, or itself wildly. This is likely because part of the neural network was weighted such that one output was always saturated which is a reasonable scenario. In later generations the best strategies still looked more or less random; however, they employed simple heuristics.

An example of a simple heuristic used was shooting in the general direction of the enemy more likely than not. Other examples include a vague strafing movement to avoid the enemy's bullets. They were also less likely to ram into the walls at full speed.

It appears that learning on Robocode on such a gross level was not feasible.

4.1.3 Amount of Diversity

A large gap between the average fitness and the maximum fitness indicates a certain amount of uncertainty. Different observed behaviours indicate that the diversity within the populations is enough to escape local optima and find and exploit new strategies.

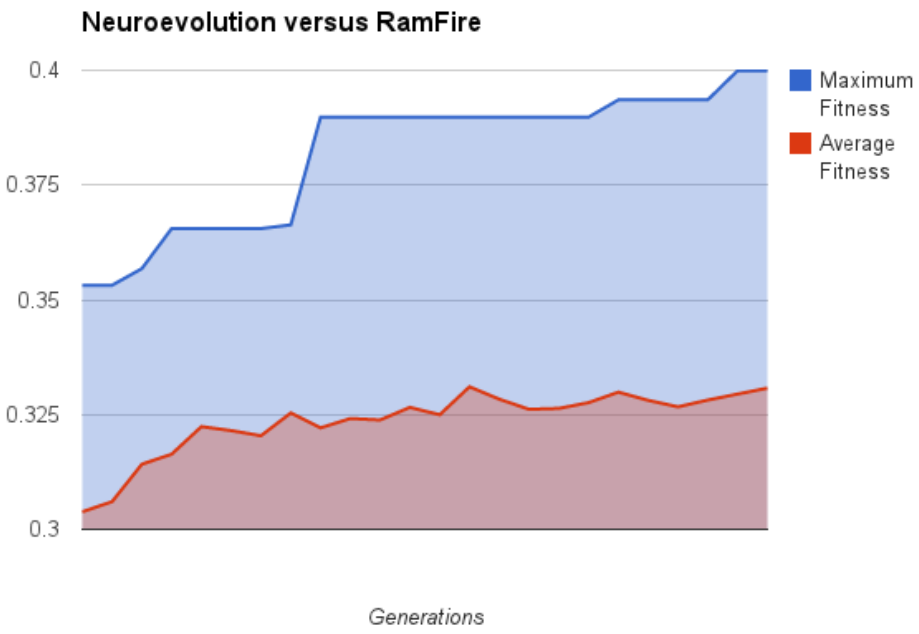

Figure 3: Learning a General Strategy using Neuroevolution against RamFire

4.1.4 Improving Performance

A larger neural network with more time and a larger population might be able to learn better behaviours.

A more refined set of inputs may also allow the robot to learn more effectively. For example, instead of simply keeping the last known enemy location we could have kept a number of past locations. This may have enabled our robot to learn the movement pattern of the enemy. This has the drawback of requiring many more inputs and weights to match.

A technique for improving the complexity would be to co-evolve useful, smaller neural networks for different modules. For example, there could be a module for each of the general actions a robot can make: moving, firing, scanning, and rotating the gun; as well as predicting the enemy's next position. The output of this final network could be used as the input to the other neural networks; avoiding the learning of a massive neural network. This is not dissimilar to a divide-and-conquer approach.

Modularizing Robocode may also allow smaller neural networks to learn more effective strategies and co-evolution may enable these to find interesting combination of techniques. We believe this would work because the effectiveness of our own ACO and genetic algorithm solutions based on selecting modular components which is the focus of the next experiment.

4.1.5 Conclusion

Overall because the fitness steadily increases as more generations are run, this experiment shows Neuroevolution may be capable of learning a good strategy to defeat the RamFire and Tracker robots. Several changes such as modularization can be made in order to improve the rate of convergence.

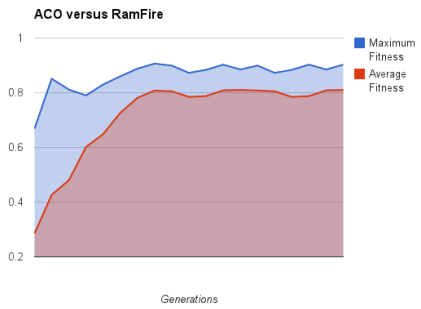

Figure 4: Best and Average Fitness of ACO against RamFire

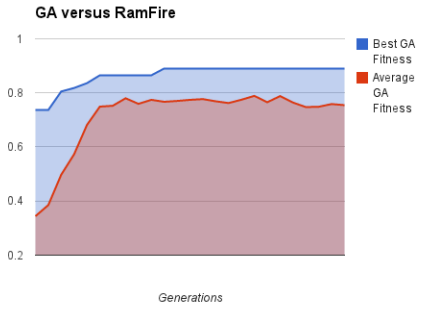

Figure 5: Best and Average Fitness of GA against RamFire

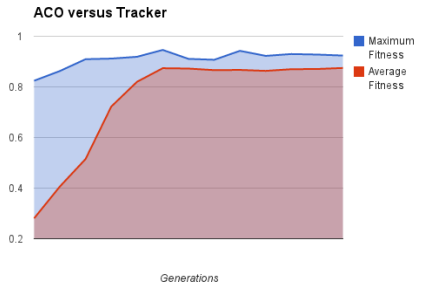

Figure 6: Best and Average Fitness of ACO against Tracker

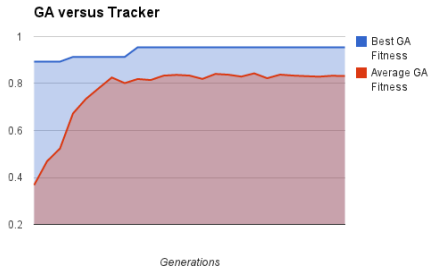

Figure 7: Best and Average Fitness of GA against Tracker

4.2 Strategy Learning using ACO and GAs

4.2.1 Rate of Convergence

The rate at which the investigated approaches converge seem to be approximately the same. The GA appears to converge faster than the ACO; however this may be due to a good random initialization or a choice of parameters which promotes early convergence.

The best fitness of both methods are approximately equal for both. Different best strategies, however, were found for the methods but are very similar. The quadruples found are described in Table 2.

4.2.2 Observed Behaviours

The best observed behaviours by fitness are described Table 2.

	RamFire	Tracker
ACO	$< 5, 1, 1, 4 >$	$< 1, 1, 2, 1 >$
GA	$< 0, 3, 4, 1 >$	$< 5, 1, 2, 1 >$

Table 2: Emergent Strategies Found using GA and ACO against the Selected Enemies RamFire and Tracker

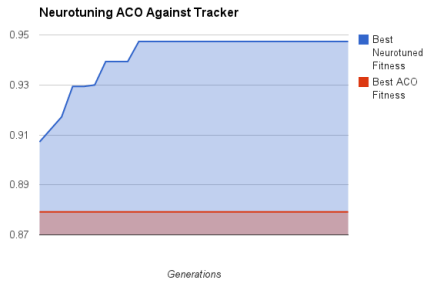

Figure 8: Neurotuning Tracking of the Best Combination Found using ACO against Tracker

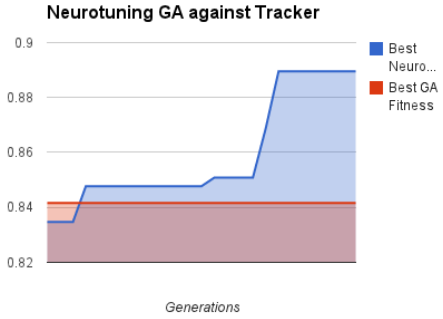

Figure 9: Neurotuning Tracking of the Best Combination Found using GA against Tracker

4.2.3 Amount of Diversity

The amount of diversity within the populations of ants and genotypes for both algorithms respectively appears to be greater within the genetic algorithm. This is evident because the average fitness is slightly lower for the GA. This means may only be due to the fact the pheromone evaporation rate is too small to increase the diversity of the ant population.

4.2.4 Neurotuning the Emergent Behaviours to Improve Performance

The fitness of the best found strategies was able to be improved using the Neurotuning methodology. The maximum reached was higher for ACO than the GA. This may also be due to random chance while calculating fitnesses.

4.2.5 Conclusion

In conclusion the two algorithms are very similar. Both were effective in learning strategies. Further, the rate of learning shown in Figures 4, 5, 6 and 7 are almost identical.

While in these examples the best fitness was found using the genetic algorithm; it may have been due to random starting positions which favoured the fitness calculation in the genetic algorithm. The fitnesses which were achieved were approximately 85-90%.

Neurotuning improved the fitness of the emergent behaviours by approximately 5-10%.

The population sizes and the number of iterations for both algorithms were the same, and therefore both methods are able to find a strategy using the same amount of resources.

5 Conclusion

5.1 Applicability of Neuroevolution to Training Robots

It was found as a result of our investigation holistic Neuroevolution was not able to learn a successful Robot function in a reasonable amount of time. We recommend extra abstractions and modularization of the code of the Robot in order to improve the training time and fitness.

5.2 Strategy Learning

Both GA and ACO seem equally proficient in combining modular strategies to win one versus one robot battles. Furthermore, Neurotuning also seems to provide some benefit against specific opponents.

No problems were encountered in the trials performed. It is expected if a robot fails to beat the enemy, then the number of pre-defined strategies must be expanded.

The GA and ACO appear to require the same resources and therefore either method is applicable to learning a combination of strategies.

5.3 Future Work

After modularizing the Neuroevolution learning, the GA and ACO can be applied to learning good combinations of the learned modular strategies.

References

[1] T. Stützle and H. H. Hoos, "Max–min ant system," *Future generation computer systems*, vol. 16, no. 8, pp. 889–914, 2000.

[2] M. Mitchell, *An Introduction to Genetic Algorithms*, ser. A Bradford book. MIT Press, 1998. [Online]. Available: http://books.google.com.au/books?id=0eznlz0TF-IC

[3] J.-H. Hong and S.-B. Cho, "Evolution of emergent behaviors for shooting game characters in robocode," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1, 2004, pp. 634–638 Vol.1.

[4] Peter, "Java math.tanh() performance," mar 2011. [Online]. Available: http://stackoverflow.com/questions/5160619/java-math-tanh-performance

[5] C. Mattiussi, P. Dürr, D. Marbach, and D. Floreano, "Beyond graphs: a new synthesis," *Journal of Computational Science*, vol. 2, no. 2, pp. 165–177, 2011.